Self-adaptive resilient service composition

Mario Henrique Cruz Torres, Tom Holvoet
Department of Computer Science
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{MarioHenrique.CruzTorres, Tom.Holvoet}@cs.kuleuven.be

***Abstract*—One aspect that permeates all large scale systems is the occurrence of failures. Continually, on any data center, failures are happening, either caused by malfunctioning disks, memories, network connections, or software bugs. Large scale failures - possibly caused be a ripple effect of smaller failures - are obviously even worse. The fact that failures are extremely hard or even impossible to predict makes them particularly challenging to cope with. A better alternative to predicting failures is creating systems that can cope with failures and autonomously adapt.**

In this paper, we investigate a decentralized self-adaptive approach to a resilient system for service composition. Our approach is based on an agent coordination mechanism known as 'delegateMAS', which is particularly suited for large-scale coordination of systems. We thoroughly evaluate this approach through large and huge scale experiments of composite services. The results from these experiments show that it is possible to create service compositions which are resilient to large scale failures.

I. Introduction

Services computing facilitates the creation of large scale applications. Services are relatively small and manageable software units with clearly defined interfaces. Applications then consist of orchestrated invocations of services, the so-called composite services. The services on which a composite service relies - called component services - have various quality of service (QoS) characteristics, such as performance, reliability, availability, accuracy. Such quality parameters can be used by a composite service to select component services when called for. Service selection and composition is particularly challenging when the system is large-scale - consisting of thousands of nodes, components and composite services - and dynamic - where QoS varies.

Particularly challenging is the situation where potentially large failures can occur. The ambition here is to create a highly resilient system for dynamic service compositions. While traditionally, a resilient system is able to deal with failures, our aim is to conceive a system that considers failures as 'business as usual', to which it gracefully molds itself.

The large or potentially huge scale of such systems (involving tens of thousands of nodes and services) makes a central selection and composition authority infeasible. In our research, we investigate a decentralized self-adaptive and self-organizing approach to dynamic service composition. In particular, we study *delegateMAS* [1], [2], a coordination mechanism originally targeted for large-scale coordination and control applications, such as traffic and logistics management, where entities need to coordinate over resources. Such coordination and control systems are intrinsically dynamic due to changes in operational (uncertainty of service time, orders, travel demand) or exceptional conditions (vehicle failures, infrastructure problems). Conceptually, the coordination mechanism also appears particularly suited for large-scale service composition.

The contributions of this paper are three-fold.

- First, we define a decentralized solution for dynamic service composition using delegate MAS. We define TaskAgents that are responsible for enacting composite service instances, and ResourceAgents that manage the usage of component services. Two delegate MASs are defined, for exploring compositions and for propagating information about intended compositions.
- Second, we thoroughly investigate the scaling of the system. We ran experiments that were large and huge in scale (up to tens of thousands of nodes and services).
- Third, we assess the behavior of the system under failing conditions, including drastic failure scenarios.

These experiments show that the approach is effective, efficient, scales linearly, and can cope even with severe failures.

This paper is structured as follows. Section II provides a technical description of the problem that this research is tackling. Section III then describes the concepts, techniques and algorithms that we employed to formulate a self-adaptive and decentralized solution to the problem. A thorough evaluation is documented in Section IV. Before we conclude, we discuss related work in Section V.

II. Problem Definition

We are interested in a system constituted by a number of *Services*and *Service Managers* interacting over a network. The system is open to new services becoming available and unavailable at any moment.

*Services*and *Service Managers* are software entities residing on the same computing node, but having distinct functionality. *Services*provide the operations that are invoked via the network, by other services. For instance, an operation can be to perform an image transformation, or to execute an algorithm. *Service Managers* on the other hand are responsible for maintaining the information about the availability, and the quality of the services which they are associated with.

Composite Services form a particular category of *Services*, which mainly work by composing the operations of other services. Composite services are made by a number of dependent activities. Each activity describes a particular operation that has to be fulfilled by a separate component service. The activities of a composite service form a graph, where

each node represents an activity and the edges represent the dataflow between the activities. Henceforth this graph implies a particular order of execution of each activity, describing serial or parallel executions, which has to be respected for the proper functioning of the composite service. The activities of a composite service are illustrated in Figure 1 An activity can only be delegated by a service compatible with that activity.

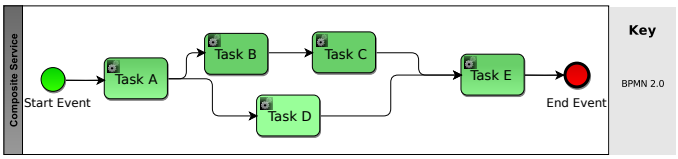

Fig. 1. A composite service is described as a graph of activities/tasks, where an activity has to be fulfilled by a service. The graph structure also implies an order of execution of each activity, possibly having serial and parallel activity executions.

We assume that there is a large number of component services providing similar operations, also referred to as *replicas*. Even being functionally equivalent, each *replica* service has varying qualities, such as a different response time, or cost, which have be evaluated at runtime.

A *Service* has a type T, called its *abstract service type*. This type is defined by the set of operations it provides. The system is open, that is, the number of *Services*and the *abstract service types* available in the system is not known a priori.

More formally, we define the system Σ, at a particular time, as $\Sigma = (S, \delta, T, M, \mu)$. S is the set of n *Service* instances $\{S_1, S_2, \ldots, S_n\}$. The function $\delta : S \rightarrow T$ maps each *Service* instance to a particular abstract service type T, where $T = \{T_1, T_2, \ldots, T_m\}, m \leq n$, is the set of abstract service types. M is the set of n `Service Manager` instances $\{M_1, M_2, \ldots, M_n\}$ and the function $\mu : M \rightarrow S$ associates one *Service Manager* with one *Service*.

The system does not have a single owner or ruler, but, instead, has several services belonging to different stakeholders. We assume the services to be cooperative in the sense that a service does not need to protect itself from other malicious services.

The system is open to services entering and leaving at any time and the number of execution requests also change constantly. As such, the number of services interacting in the system or service requests is unknown. Also the various sources of dynamism are unknown in advance. Several problems can happen at any moment, services may fail, the network can become unresponsive or even partitioned, and service execution requests can arrive in spikes.

The node failure model is the *crash-recovery failure model* [3], [4]. Nodes can fail by crashing or losing their network connection, and may later recover. We assume that nodes do not arbitrarily misbehave, that is, they do not generate wrong data or send faulty messages to other nodes, such as what can happen in Byzantine systems [5].

We do not consider partial node failures, such as a service failing but its service manager staying alive. When a node fails, the service and service manager operating at that node both fail. Composite services only fail if they can not find all needed component services, within a given timeout. From the point of view of the system, we distinguish two types of failure situations, according to their scale, which is defined by the number of failed services:

- Few inaccessible services. Less than 20% of the services of the system are broken.
- Several inaccessible services. More than 80% of the services of the system are broken.

The first type of failure situation only disrupts a small percentage of services and, consequently, a small percentage of composite services which rely on the broken services. The second type of failures can cause system wide failures, which may be rare but can disrupt several services at once.

We model the failures in our system by stochastically removing nodes from the service network using different probability distributions. We model node failures using an **exponential** probability distribution. The exponential distribution is memory-less, meaning that given a certain constant failure rate, the occurrence of a failure does not affect the probability of other failures also happening [6]. The duration of a failure is modeled using a **Poisson** distribution, having a very small probability of a service not recovering from a crash.

A failure can be seen as a function $\epsilon : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ which takes a set O of services, where $O = \{S_\alpha, S_\beta, \ldots, S_\gamma\}$, $O \in \mathcal{P}(S)$ and returns another set $O' \in \mathcal{P}(S)$. Additionally, the function ϵ randomly removes elements from the input set O, adding them, with a probability ρ according to the exponential probability distribution, to the set O', where $O' \subseteq O$.

We model very large failures, which mainly are failures disabling several services at once, using an **exponential** distribution. In our model there is a very small probability to have said system wide failures. However if such failure happens, a large number of services stop working, having a great impact not only on the composite services using such failed services, but also on the entire system.

Very large failures disrupt at least 80 % of services on the system at a given time. More formally, a very large failure can be modeled as a function $\epsilon' : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, which takes a set O of services, where $O \in \mathcal{P}(S)$, and returns a set $O' \subset O$, where $\|O'\| \leq (1-\rho) * \|O\|$. The set O' is created by randomly removing members from the input set O, with a probability $\rho = 0.8$ according to the exponential probability distribution.

Besides removing elements from the set of services S, participating in the system Σ, we also model the duration of each service's failure. A failure duration is modeled by the function $\varphi : S \rightarrow \mathbb{R}$ which takes a service $S_i \in S$ and returns a duration in seconds, according to the **Poisson** distribution, with a given mean β.

The challenge is to create a distributed mechanism that allows composite services operating on a large network of services, remain working, or degrade gracefully even in the presence of large scale failures.

III. Composite Services Coordination Using DelegateMAS

The main objective of coordinating the actions of different services in a service network is to ensure that the services are

enacted in an effective and efficient manner, especially in the presence of failures. Coordination of dynamic service compositions can be achieved by, for instance, enforcing contracts between the services, establishing conventions of operation, or creating control hierarchies that can impose restrictions on each service operations.

Our approach, however, relies on a decentralized decision making process, named *delegateMAS*. In *delegateMAS*, information about the services participating in the service network is spread around the network. Each service participating in the *delegateMAS* system takes individual decisions based on information that it can retrieve from the system. *delegateMAS* allows creating and adapting robust plans on a distributed environment by having services share their short term intentions. That way, services can adjust their plans accordingly, taking others' intentions into account. A system global solution then emerges from the actions of each individual service participating in the system.

The two basic agent abstractions in *delegateMAS* are the *TaskAgents* and *ResourceAgents*. In our approach, *TaskAgents* are responsible for assigning good quality component services to the activities needed by a composite service. *TaskAgents* continually monitor the network, looking for better alternatives to the currently selected services. *ResourceAgents* represent the service managers in the system. Each *ResourceAgent* is responsible for bookkeeping information regarding a service residing in the same computing node. They communicate with their associated service to, for instance, inspect how loaded they are, and to create short term forecasts of their future load.

These agents delegate work to specialized light-weight agents that perform specific, dedicated operations on their behalf. *ExplorationAnts*, *IntentionAnts*, and *FeasibilityAnts* are the specialized agents used in *delegateMAS*. *ExplorationAnts* are responsible for finding component services that can be available to execute tasks of a composite service. *IntentionAnts* are responsible for indicating to *ResourceAgents* that a *TaskAgent* intends to use their operations. *FeasibilityAnts* are used by *ResourceAgents* to spread *QoS* information on the environment. Hence, a *TaskAgent* does not directly contact other *ResourceAgents*, but instead, delegates the job of finding good quality *ResourceAgents* to a *ExplorationAnt* agent. Figure 2 illustrates the abstractions used in a *delegateMAS* solution.

We explain, in the following sections, each agent type and how we achieve a coordinated usage of resources by these agents and the trade-offs involved in designing a *delegateMAS* system.

A. TaskAgents

TaskAgents are responsible for finding the best component services to fulfill the activities described by the composite service it represents, at a given time. They are also responsible for keeping track of the execution of a composition and respecting the desired qualities specified by the composite service. They are essentially following the *BDI* agent architecture. *BDI* agents continually monitor the environment, creating their (beliefs), consider possible options on how to proceed (desires), and choose a particular option as their intention. They then create action plans to fulfill their intention. That way, *BDI* agents are designed to cope with environments subject to

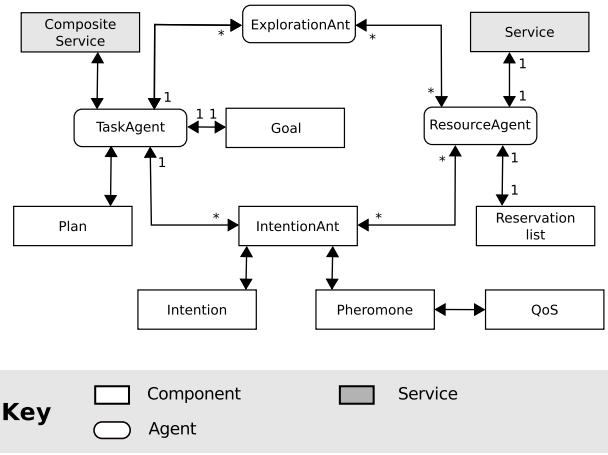

Fig. 2. A *delegateMAS* adapted to work on the service composite service coordination problem. *TaskAgents* are responsible for the proper invocation flow of component services. *ResourceAgents* represent component services, and are also responsible for bookkeeping reservations to use them. The information about the quality of a particular component service is spread via pheromones on the system.

sudden changes and failures in a pragmatic, computationally manageable way [7].

Each *TaskAgent* represents one composite service instance, and has the following main responsibilities:

- receive composite service description as input
- receive a *Service Level Agreement* specifying the desired qualities for its composite service;
- select possible component services to provide operations to its service composition instance, creating what we call a plan of execution;
- inform the *ResourceAgent* from the selected component services that it intends to use their services
- keep inspecting the network for possible alternative component services, while its service composition is being executed;
- evaluate if there are better component services to use in the composition, and decide to use them or not, which is determined by the *TaskAgent* commitment strategy;
- upon the completion of is service composition instance, spread information about the quality of the engaged component services.

The exploration of component services is delegated to *ExplorationAnt*. The intention maintenance is delegated to *IntentionAnt*.

An important aspect of *TaskAgent* regards which commitment strategy is used. As studied in [8] , commitment strategies impact the overall functioning of a *Multiagent Systems* system. Hence, we will explain in further details possible commitment strategies to be used by the *TaskAgent*.

Information about *TaskAgent* intentions is distributed over the environment by sending out *IntentionAnts* which drop

information - called pheromones - at *ResourceAgents*. The pheromones represent the quality of a particular component service. A pheromone contains a tuple of quality parameters, which are domain specific, such as the *response time, price, availability* of a component service. This pheromone information is used by *ExplorationAnts*. We explain the behavior of both *ExplorationAnts* and *IntentionAnts* below.

1) ExplorationAnts: ExplorationAnts are issued by *TaskAgents* for exploring alternative solutions that maximize a *TaskAgent* goal. For instance, an *ExplorationAnt* willing to minimize the time to execute a service composition tries to find component services which are fast and that are not overbooked. After selecting *ResourceAgents* that could participate in the service composition, the *ExplorationAnt* reports back its findings to their originating *TaskAgent*.

An *ExplorationAnt* explores a possible solution by recreating the execution of a composite service, searching for the right type of *ResourceAgents* needed for each activity of their composite service description. In order to know which *ResourceAgents* a *ExplorationAnt* should look for, *ExplorationAnts* are instantiated with a graph representing the service composition. Having this graph at hand, the *ExplorationAnts* start crawling the network searching for suitable component services. The *ExplorationAnts* then check the *QoS* and *pheromone* levels leading to a particular *ResourceAgent*, and decide if they should keep searching for other *ResourceAgents*, or if they should select the *ResourceAgent* they just found.

A naive exploration strategy would be to flood the network request the *QoS* and *pheromone* of each possible *ResourceAgent*. However that would be computationally costly and time consuming, especially in large scale systems. *ExplorationAnts* use an *Ant Colony Optimization* approach to explore the possible *ResourceAgents*, without flooding the network. An *ExplorationAnt* tries to find a path containing *ResourceAgents* capable of performing the operations needed by its originating *TaskAgent*. *ExplorationAnts* take all *QoS* parameters into account before commit to use a particular *ResourceAgent*. *ExplorationAnts* evaluate the quality of a particular *ResourceAgent* using an heuristic η, defined below:

$$\eta : (q_1, \ldots, q_n) \rightarrow \mathbb{R}, \text{ such that}$$
$$\eta((q_1, \ldots, q_n)) = \frac{1}{\sum_i^n q_i^*}, q_i^* \text{ is the normalized } q_i$$

where η is evaluated every time an *ExplorationAnt* checks for the quality of a service provided by a given *ResourceAgent*. An *ExplorationAnt* decides on which path to follow according to the probability:

$$P_{ij}(t) = \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}(t)]^\beta}{\sum_{l \in N_i} [\tau_{il}(t)]^\alpha [\eta_{il}(t)]^\beta} \quad (1)$$

where τ is the pheromone level, α indicates how worth is the pheromone information to the *ExplorationAnt*, η is the heuristic that takes the *QoS* into account, and β indicates how worth is this quality information to the *ExplorationAnt*.

ExplorationAnts select a path to follow according to the probabilities defined in Equation 1. Thus there is always a high probability for the exploratory behavior, unless the parameter α is set to high. If α is set to high, *ExplorationAnts* will only follow the paths with the highest pheromone concentration. An *ExplorationAnt* returns back its findings to its origin *TaskAgent* when, either they reach their *time to live*, or they have found a valid path, that is, they have found enough *ResourceAgents* to fulfill the needs of their originating *TaskAgent*.

A simplified view of the behaviour of an *ExplorationAnt* is illustrated in the algorithm 1.

Algorithm 1 Exploration behavior from an ExplorationAnt

```
α ← initialize
β ← initialize
workflow ← composite service graph
while ∃workflow activities without component service ∨ timed − out do
    currentactivity ← next workflow activity
    retrieve QoS and Pheromones from neighbors
    select component service according to P(α, β)
    currentactivity ← selected component service
end while
if all workflow activities have an assigned component service then
    return workflow
else
    return failure to find component services
end if
```

2) IntentionAnts: IntentionAnts are responsible for communicating the intentions of a *TaskAgent* to use a set of *ResourceAgents*. *IntentionAnts* inform each selected *ResourceAgent* to participate in the service composition about the intention of a *TaskAgent* to use the service of the *ResourceAgent*, at a particular time. *IntentionAnts* also keep track of each *ResourceAgent*'s performance. An *IntentionAnt* communicates back with its *TaskAgent* to inform about any changes on the expected performance of the execution of a component service on that particular path. That way, the *TaskAgent* can decide to look for alternative component services and change its intentions to better ones.

B. Resource Agents

A *ResourceAgent* is responsible for a single component service, and they provide the quality information from the service they are bound to. In our model, the quality information is represented using a vector $q = (q_1, \ldots, q_n)$, where $q_i \in \mathbb{R}$, where each component q_i corresponds to a desired quality, e.g. `ResponseTime, Price, etc..`

They further (1) manage the bookkeeping of the future usage of component services, i.e. their schedule, and (2) use the information brought by *IntentionAnts*, to make predictions of their future load. Every time an *ExplorationAnt* wants to know about the future availability of a component service, it asks this information to the corresponding *ResourceAgent*. The intention information that is left behind by *IntentionAnts* allows the *ResourceAgent* to make accurate an prediction of the expected short-term forecast qualities of the service.

IV. Evaluation

To assess the approach and its behaviour on a large scale service network, we have implemented all abstractions from our *delegateMAS* model, along with support for two distinct service categories, composite services and component services. The prototype uses techniques of both emulation and simulation, and is in fact deployed as a large scale distributed system.

The system is composed by the composite and component services. Composite services, as explained before, delegate their operations to component services. Component services, on the other hand, simulate the execution of a certain operation requested by other services. The component services are also real services, but instead of doing real calculations, they simulate the time it would take to do a calculation. When a component service receives a request, it randomly generates a number around its configured average processing time and stays in a busy state until the 'completion' of its job.

The system was deployed on a computer cluster of 32 compute nodes. Each compute node has 96 GB of RAM and two sockets, each with a 6 core Intel Xeon X5660 (Westmere-EP) processor. Using hyper-threading each node therefore can use up to 24 hardware threads. The node's operating system was a Linux with kernel 3.2.0-52-generic SMP. Nodes are connected via InfiniBand network in a star topology, having an average round-trip time between any two nodes of 0.131 ms.

Our evaluation consists of generating different failure patterns and analysing how the system reacts to such failures. We also assess how our algorithms compare to a purely *Reactive* solution in the case of massive failure.

A. Purely Reactive Service Composition

We created a purely reactive service composition mechanism to compare to our *delegateMAS* solution. The reactive algorithm is executed by the *ExplorationAnts* every time a *TaskAgent* instantiates a new service composition. When a *TaskAgent* initiates a service composition instance, it iterates over the activities of the composite service and creates an *ExplorationAnt* that searches the network for a single activity. When this reactive *ExplorationAnt* finds at least 5 component services capable of performing the desired activity, the *ExplorationAnt* evaluates their *QoS*, selects the best one and returns its address to the *TaskAgent*. After the successful completion of an activity, the *TaskAgent* creates a new *ExplorationAnt*, but this time for the next activity of the composite service. The utility function the reactive *ExplorationAnts* use is the same as the utility function that the *delegateMAS ExplorationAnts* use to evaluate the *QoS* of a set of component services.

The *TaskAgent* never makes a plan of which component service it will use in the future, it always looks for new component services when it needs them.

B. Scenario

We are interested in failures in large scale systems, so we perform our evaluation in two systems having 16,000 (16k), and 64,000 (64k) services deployed in the computer cluster. Composite services either have two or three sequential tasks to perform. Component services, on the other hand, offer services of three types, **A, B, or C**. The number of services used in our experiments is detailed in Table I.

TABLE I. Characteristics of the Services used in our experiments

16,001 Services				
Service Type	Nb Services	%	Mean Exec.Time (s)	StdDev (s)
Composite 2	1029	6.43	–	–
Composite 3	494	3.08	–	–
Factory A	4715	29.46	20.03	5.82
Factory B	4762	29.75	19.91	5.79
Factory C	5002	31.25	19.90	5.77
64,002 Services				
Service Type	Nb Services	%	Mean E.Time(s)	StdDev (s)
Composite 2	4327	6.76	–	–
Composite 3	2129	03.32	–	–
Factory A	19228	30.04	20.02	5.82
Factory B	19270	30.10	19.99	5.78
Factory C	19047	29.76	20.02	5.73

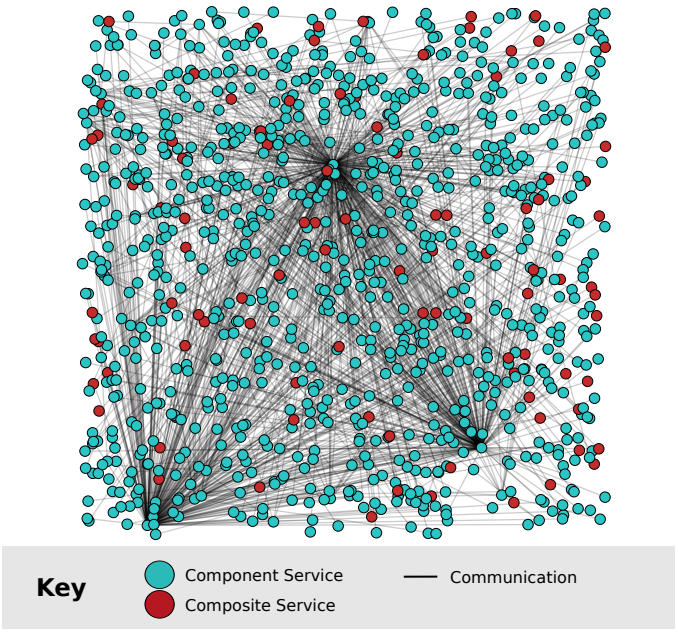

Fig. 3. Each node in the graph represents a service. There are two service categories, composite and component services. The edges represent to which other services, a service is directly connected.

Each service is only aware of the presence of a small number of other services, given by the topology of the network. In order to make our experiments more realistic, we used internet topology data collected by the CAIDA project. The CAIDA Skitter project created an internet topology graph after running traceroutes from scattered sources to millions of internet hosts, in the year of 2005 [9]. Figure 3 shows a sample network containing 1000 nodes created using the Skitter data. We use this topology data to indicate to each service in our system, which other services they are directly connected to.

Table II provides an overview of the networks we have used in our experiments. It is important to note that both networks were created based on data from the CAIDA project, and as such have very similar properties.

We configure the algorithms *delegateMAS* and the *Purely*

TABLE II. SERVICE NETWORK METRICS

Metric	Value	Value
Nodes	16,001	64,001
Edges	18,752	103,101
Diameter	5	6
Radius	3	3
Average path length	3.74	7.25
Number of shortest paths	256016000	2147483647
Average Degree	2.344	3.222
Modularity	0.779	0.749
Number of Communities	33	14
Average Clustering Coefficient	0.271	0.1
Total triangles	942	12014

Reactive Service Composition with the same timeouts to explore the network. The timeouts for exploration is set to 1.8 s. The *delegateMAS* algorithm is configured with $\alpha = 0.5$ and $\beta = 5$, (eq. 1), what means that it gives more preference to already known component services (represented by the pheromone information), than to the promised quality offered by a new component service (heuristic information).

C. Failures

We are interested in the behaviour of the system during the occurrence of failures. Initially we let the system execute for 10 seconds, then we start sending failure messages to the services in the system. When a service receives a "Failure" message, it turns itself down for the time indicated in the message. When a service receives a "Failure" message it stays on average 30 seconds in a fail state.

We are also interested in the effects of failures in large scale and very large scale systems. In a small scale failure, for each service on the system, we generate a random number between [0,1] using the uniform probability distribution, if the number is smaller than or equal 0.20, we send a "Failure" message to that service. For the large scale failures, we follow the same procedure, but send a "Failure" message if the generated number is smaller than or equal to 0.8.

D. Experimental Results

We are interested in how well the *Reactive* dynamic service composition and the *delegateMAS* service composition work in a large scale network with failures. In order to measure how the system performs, we mainly focus on one metric, which is the duration of the composition execution. By *duration of the composition execution*, or simply *composition times*, we mean the time between the instantiation of a service composition, the time spent searching for available component services, and the time for each component service to execute the task at hand. We performed each experiment at least 10 times, in order to have statistically relevant results.

An important aspect of creating service compositions is to guarantee that the system operates properly during the entire execution of the system. However the system behaviour may be hidden if only analyzed using statistical summaries, such as the average duration of composition times. We are interested in many aspects regarding the system behaviour, when services suddenly fail. We also want to investigate how good a service composition is in relation to the remaining service compositions, for instance, measuring the standard deviation of the composition execution times, in the presence of failures.

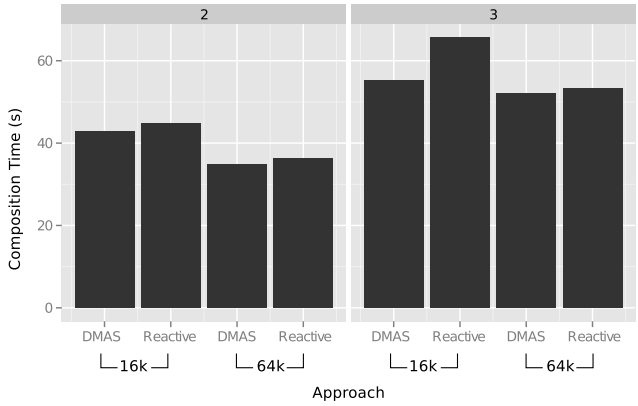

Fig. 4. Comparison of the average composition times of service compositions created using *delegateMAS* or the Reactive approach. The service compositions are made of 2 or 3 tasks, and the system was evaluated with a 20 % component service's failure rate. *delegateMAS* selects component services which produce a better (shorter) composition time for the different network size and composition types.

We begin describing the system behaviour when it is subject to a small scale failure, that is, 20% of the nodes are failing. We describe the system for a service network having 16k services and for a bigger network, having 64k services.

Directly comparing both approaches, we can see that a purely *Reactive* approach creates compositions with higher average composition times than the *delegateMAS* approach. Figure 4 shows a comparison of the averages of the composition times created by the two different approaches, in networks having 16k and 64k services. It is interesting to note the difference in the average composition times for composite services having 2 or 3 tasks. In a network having 16k services, *delegateMAS* is 4.55 % better than the *Reactive* approach when the composite service has only 2 tasks. On the other hand, *delegateMAS* is 24.15 % better than the *Reactive* approach when the composite service has 3 tasks.

The behaviour of both algorithms under 20 % failures in a very large network, having 64k services is not as performant as in a smaller network. The reactive approach for compositions having 2 tasks was only 2.36 % slower than *delegateMAS*. Again, in a network having 64k services, *delegateMAS* performed better for compositions having 3 tasks. In compositions with 3 tasks, the *Reactive* approach was 4.6 % slower than *delegateMAS*.

Figures 5 and 6 show the normalized distribution of the composition times of the created service compositions, in a scenario with 20% of nodes failing and 16k services, for the *Reactive* and *delegateMAS* approaches respectively. It is possible to see that the distribution of the average composition times in the *delegateMAS* approach is more concentrated around the mean, than the purely reactive approach. Another interesting aspect is how *delegateMAS* is more affected by the different number of tasks in a composite service. In *delegateMAS* the distribution of composition times around the mean for composite services having 3 tasks is larger than for composite services with only 2 tasks, as illustrated in Figure 6. We believe this difference is due to the fact that composite services with more tasks may be more affected in the presence

of failures, since they rely on more component services.

Networks having 64k services and 20% failure have a distribution of composition times which is very similar to the distributions for networks having 16k services. Hence, we do not show these results here.

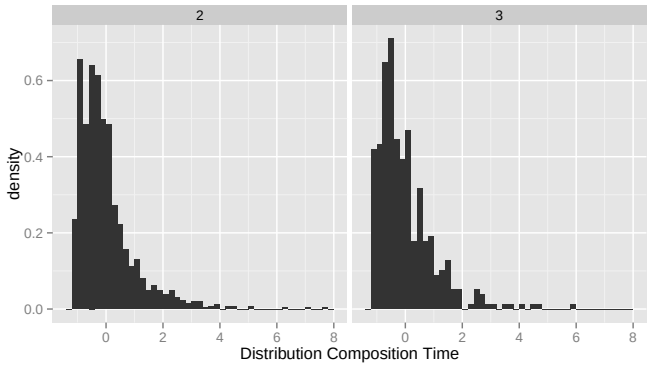

Fig. 5. Normalized mean composition time using a Reactive algorithm, 16k nodes, 20% failures.

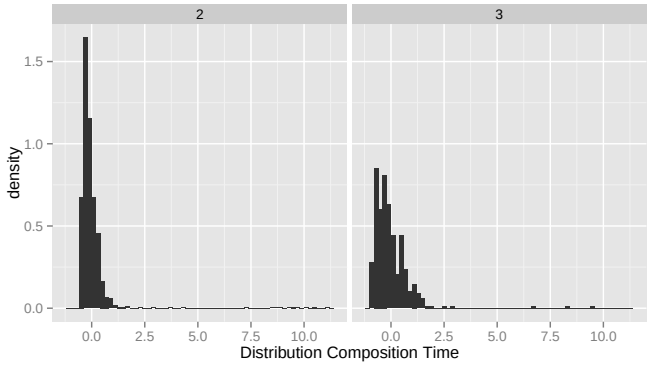

Fig. 6. Normalized mean composition using *delegateMAS*, 16k nodes, 20% failures.

We are also interested in the behaviour of the system during its whole execution. The service compositions times should, as much as possible, have a small standard deviation of composition times. Figure 7 shows the system behaviour during its whole execution, using the reactive approach, when facing 20% failures. In a system using the *Reactive* approach, having 16k services and 20 % failures, the standard deviation of the service compositions times was large, what can be seen by the presence of many outliers.

On the other hand, Figure 8, shows the behaviour of the system, having 16k services and 20% failures, using the *delegateMAS* approach which has a smaller standard deviation for the composition times, indicated by the presence of less outliers. We believe the difference in behaviour stems from the "memory" properties brought by the pheromones that each *ExplorationAnt* deposits in the environment, helping to guide other agents to use good component services.

A shortcoming of our approach is that it produces bad service compositions (large composition time) when the system still does not posses enough information about good component services, what can be seen at the left hand side of Figure 8.

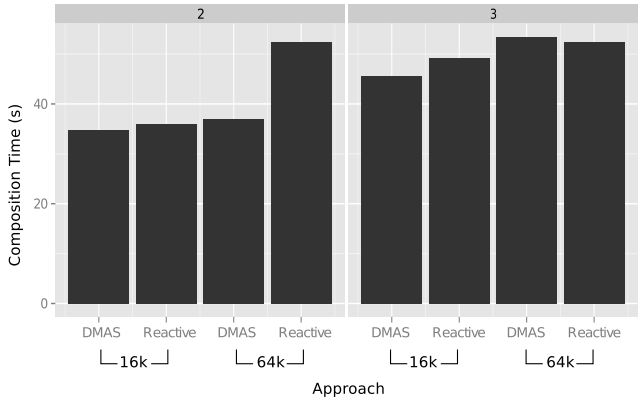

Fig. 9. Comparison of the average composition times of service compositions created using *delegateMAS* or the Reactive approach, in a system with 80% failure rate. As in the experiment with 20% failures, service compositions are made of 2 or 3 tasks. The average composition time difference between the two approaches is not so accentuated in this scenario, but *delegateMAS* still manages to create the best compositions.

From the collected data, we can conclude that *delegateMAS* provides superior performance and is less affected by small failures (20%) than a purely reactive approach. *delegateMAS* works particularly well on moderately large service networks (16k services), and is slightly better than the *Reactive* approach on a very large network (64k services).

When the system is hit with large scale failures, that is, more than 80% of the services stop working, most part of composite services can not find suitable component services anymore, timing-out. However, the compositions which still manage to find suitable component services are not really affected by the failures.

Figure 9 shows a comparison of the averages of the composition times created by the *delegateMAS* and *Reactive* approaches, in networks having 16k and 64k services and 80% failure rate. In a network having 16k services, *delegateMAS* is 3.9% better than the *Reactive* approach when the composite service has only 2 tasks. Not like in the 20% failure rate scenario, *delegateMAS* is only 8.1% better than the *Reactive* approach when the composite service has 3 tasks. In networks having 64k services, the reactive approach for compositions having 2 tasks was 41% slower than *delegateMAS*. Again, in a network having 64k services, *delegateMAS* performed better for compositions having 3 tasks. In compositions with 3 tasks, there was no significant difference between the approaches. A interesting result was that *delegateMAS* 1.9% slower than the *Reactive* approach in a very large network with 80% failures.

A interesting aspect of very large scale failures in our experiments is that Composite services which manage to find suitable component services can still suffer a great variation in the quality of their compositions, what happens with both *delegateMAS* and *Reactive* approaches. Large scale failures are very difficult to cope with, even our approach which performed particularly well on a system having 20% failure rate, becomes unstable when the system has a 80% failures rate. Figure 10

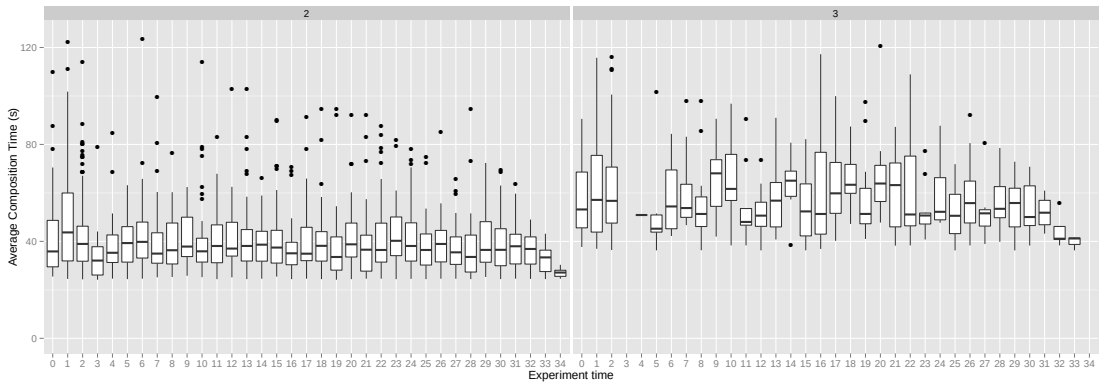

Fig. 7. Reactive approach. Average composition times over the execution of the system with 16k services, under a 20% failure. The box-plots show a large standard deviation for the compositions, over time, what can be seen by the large number of outliers.

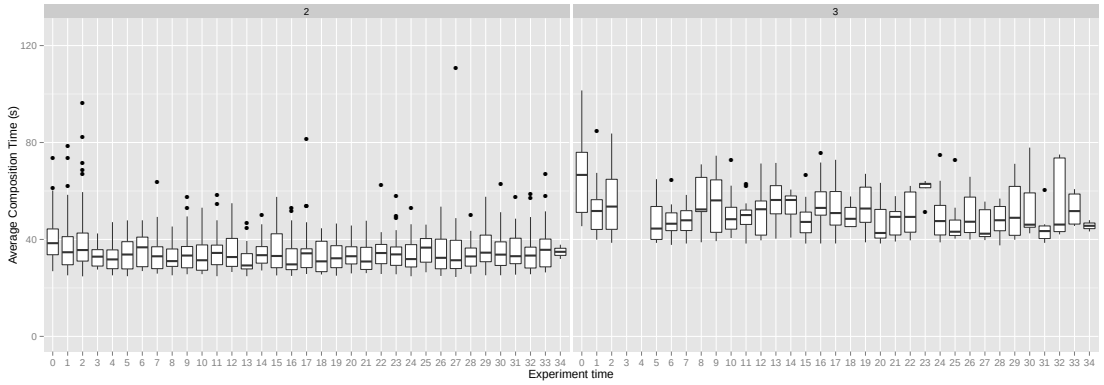

Fig. 8. *delegateMAS* approach. Average composition times over time, during the execution of the system with 16k services, under a 20% failure. The box-plots show that over the duration the experiment, *delegateMAS* managed to maintain a small standard deviation for the composition times.

shows an execution of the system with 64k services, and 80% failure rate, using the *delegateMAS* approach. We can see that many times no compositions are executed. That is due to the large number of component services unavailable at any moment.

V. Related Work

The dynamic service selection and composition problem has different challenging aspects, originating from the unpredictability of environments, the scale of the systems, different goals of each service, etc. There are different techniques to create dynamic service compositions, however, normally, each technique focuses on one particular aspect of the service compositions, such as resilience to failures, composition speed, *QoS* guarantees, optimization of quality parameters, decentralized or centralized compositions, to cite a few [10].

Broker based archictectures in principle require centralized or minimally replicated brokers responsible for answering queries about the *QoS* of component services that can be used to create dynamic service compositions. Cardellini et.al. [11] proposes a broker based architecture, where a central broker, or a cluster of replicated brokers, maintains *QoS* information about all the component services in the system. The main difference between Cardellini approach and ours is that we propose a completely decentralized architecture, what is accomplished via spreading information in all services participating in the system.

Ma et.al. proposes to not only select component services based on their current *QoS* attributes, but, instead, to reconfigure the component services, in order to improve the *QoS* of the composed services as well [12]. The work also proposes to use UDDI (Universal Description Discovery and Integration) repositories to get *QoS* information about the available services, what resembles a centralized solution. Our approach does not assume a service repository, as the UDDI, and does not consider reconfigurable services as well. We focus on runtime adaptation of the composite services, and not so moch on the optimality of the possible compositions.

QoSMOS is a framework to create service oriented systems, focusing on achieving pre-defined *QoS* requirements [13]. QoSMOS combines different techniques to allow the specification and execution of services which adhere to the desired *QoS* requirements. It uses Markov models to determine the reliability and performance of different quality parameters of component services. Our solution does not maintain a statistical model regarding the component services of the system, since we focus on large scale systems, this model could become infeasible to maintain.

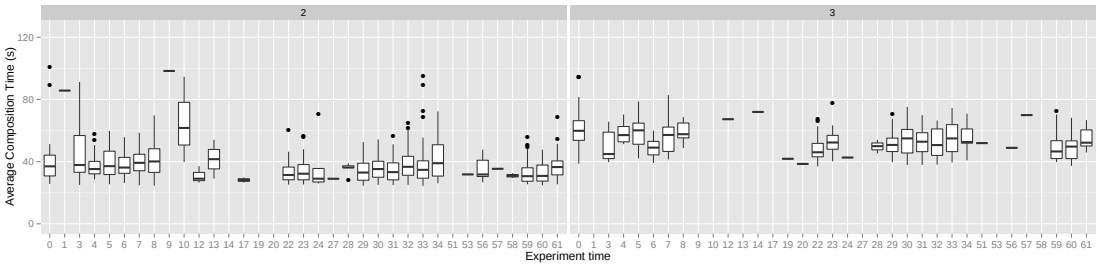

Fig. 10. Overview of the composition times, when the system has an 80% failure rate. The system uses *delegateMAS* approach to find,select, and compose from a pool of 64k services. The box-plots show a large standard deviation for the compositions, over time, what can be seen by the large number of outliers.

Mostafa et. al. presents an approach that shares many concepts with our approach [14]. It presents a decentralized composition mechanism based on stigmergy properties, regarding the quality of the used services. It leaves open how one could architect such decentralized mechanism, and how the algorithms react to large scale failures. The main difference with our approach is that we propose to use active agents, the *ResourceAgents*, to maintain information about the component services, and to do short term load predictions, facilitante the coordination between the several composite services.

Philipp Leitner *et al.* [15] proposed a framework called `PREvent`, which is a system that integrates monitoring, prediction, and adaptation of service compositions. The main goal of the `PREvent` framework is to adapt service compositions in order to prevent *Service Level Agreement* violations.

The framework mainly consists of three components: Composition Monitor, *Service Level Objective* Predictor, and the Composition Adaptor. The Composition Monitor is responsible for monitoring the runtime data. Prediction of violations are handled by the *Service Level Objective* Predictor, which uses learning techniques to identify the services that can cause *Service Level Agreement* violations in the future. Finally the Composition Adaptor component is responsible for identifying and applying adaptation actions. Our approach does not use learning techniques, but instead spread information on the environment to be able to do a probabilistic search on it.

The work on Stein et. al. explicitly focuses on creating robust service compositions [16]. The work uses decision theory for dealing with the uncertainty associated with component service providers. It proposes a mechanism for component service selection that explicitly takes the reliability of the created composition into account. The approach uses service redundancy for critical tasks of the composite service, what they call a workflow, in order to achieve a higher reliability of the composite services. The main difference with our approach is that we are interested in large scale networks, and not the optimality of composition, while Stein et. al. approach is interested in guaranteeing a near optimal allocation of the component services that will participate in the composition.

Another related area are studies which try to preemptively predict the *QoS* of services and service compositions [17].

A very interesting approach for service selection and composition is proposed by Klein et. al. [18]. Klein et. al. approach the problem of service selection and composition in cloud environments.

The work makes a distinction between the *QoS* parameters of the services and the *QoS* parameters of the network. This process allows the approach to select services with a low latency between the client and the component services. After selecting candidate services which have a low latency between them and the composite service, the approach realies on genetic algorithms to select the best available component services, based on their *QoS* parameters.

The architecture of our approach is very different than the solution proposed by Klein et. al. We propose to have active entities at the component service and composite service sides, which have to communicate in order to create a good allocation for which component service will execute which activities from the composite serivces. Our approach uses pheromones which are spread over the network, to keep information about the past quality of component services, while Klein et.al. uses genetic algorithms to improve the service selection algorithm.

We believe the main difference between our approach and the state-of-the-art approaches we describe in this section of the paper, is that oru approach focus on creating a system that can still operate in the face of large scale failures. We propose an architecture and a mechanism on how to enable the dynamic composition of services in a way that incorporates failure at its design.

VI. Conclusion

Very large distributed service systems are becoming more and more ubiquitous in nowadays computing environments. More applications are created to be executed on cloud datacenters. Such applications rely on the usage of dedicated services, which are scattered around the network. Large applications continually face the presence of failures which can be caused by a myriad of causes, such as malfunctioning disks, memories, network connections, or software bugs.

A particularly challenging aspect about failures is that it is extremely hard or even impossible to predict them. We believe that any large system design should embrace failures as business as usual.

In this paper we have proposed a decentralized mechanism, called *delegateMAS*, which helps one to create applications, in the form of composite services, which are resilient to failures. We compare our approach to a purely reactive approach in terms of the quality of the service compositions each approach can create when facing large scale system failures. We show that *delegateMAS* is better at creating more stable service

compositions, and selecting the best component services for a network having 16,000 services. In networks having 16k services and with a 20% failure rate, the alternative approach, called *Reactive* approach, selected component services for its compositions having 2 or 3 tasks which were, on average, 4.55% and 25.15% slower than our approach.

We are aware that our approach, *delegateMAS*, performed better in the tested scenarios, but that it also has other costs, such as communication costs, which were not explored in this paper.

A improvement that needs to be done in our approach is that certain parameters, for instance, the exploratory and heuristic behaviour parameters, have to be fine tuned before using the algorithms. We believe it would be better to create a self-adaptive layer that can help fine-tuning the algorithms.

As future work, we plan to investigate how to adapt *delegateMAS*, in order to maintain the good quality service composition execution time predictions, while lowering the composition time as well.

Acknowledgments

Test runs of the presented algorithms were run at the Exa-Science Life Lab, Leuven, Belgium. This research is partially funded by the Research Fund KU Leuven. This research is also supported by the IWT project DiCoMas (060837/SBO), GOA/2011/004, and iMinds.

References

[1] R. Claes, T. Holvoet, and D. Weyns, "A decentralized approach for anticipatory vehicle routing using delegate multiagent systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 12, no. 2, pp. 364–373, Mar. 2011, ISSN: 1524-9050. DOI: http://dx.doi.org/10.1109/TITS.2011.2105867. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/309776.

[2] T. Holvoet and P. Valckenaers, "Beliefs, desires and intentions through the environment," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM, 2006, pp. 1052–1054, ISBN: 1-59593-303-4. DOI: 10.1145/1160633.1160820. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/134283.

[3] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," English, *Distributed Computing*, vol. 13, no. 2, pp. 99–125, 2000, ISSN: 0178-2770. DOI: 10.1007/s004460050070. [Online]. Available: http://dx.doi.org/10.1007/s004460050070.

[4] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *Software Engineering, IEEE Transactions on*, vol. SE-9, no. 3, pp. 219–228, 1983, ISSN: 0098-5589. DOI: 10.1109/TSE.1983.236608.

[5] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *Computer Safety, Reliability, and Security*, Springer, 2003, pp. 235–248.

[6] R. Vitenberg, D. Zinenko, K. Kvilekval, and A. Singh, "Analyzing performance of lease-based schemes under failures," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, 2011, pp. 193 –202. DOI: 10.1109/SRDS.2011.31.

[7] A. S. Rao and M. P. Georgeff, "Bdi agents: from theory to practice," in *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, L. Victor and G. Les, Eds., Cambridge, MA, USA: MIT Press, 1995, pp. 312–319.

[8] D. Kinny and M. Georgeff, *Commitment and effectiveness of situated agents*. Dept. of Computer Science, University of Melbourne, 1991.

[9] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," pp. 177–187, 2005.

[10] S. Dustdar and W. Schreiner, "A survey on web services composition," *International Journal of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.

[11] V. Cardellini and S. Iannucci, "Designing a broker for qos-driven runtime adaptation of soa applications," in *Web Services (ICWS), 2010 IEEE International Conference on*, 2010, pp. 504 –511. DOI: 10.1109/ICWS.2010.77.

[12] H. Ma, F. Bastani, I. Yen, and H. Mei, "Qos-driven service composition with reconfigurable services," *Services Computing, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2011, ISSN: 1939-1374. DOI: 10.1109/TSC.2011.21.

[13] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 387 –409, May 2011, ISSN: 0098-5589. DOI: 10.1109/TSE.2010.92.

[14] A. Mostafa, M. Zhang, and Q. Bai, *Trustworthy stigmergic service composition and adaptation in decentralized environments*, 2014. DOI: 10.1109/TSC.2014.2298873.

[15] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "Monitoring, prediction and prevention of sla violations in composite services," in *2010 IEEE International Conference on Web Services*, 2010.

[16] S. Stein, T. Payne, and N. Jennings, "Robust execution of service workflows using redundancy and advance reservations," *Services Computing, IEEE Transactions on*, vol. 4, no. 2, pp. 125 –139, Feb. 2011.

[17] K. Geeblen, "Qos prediction for web service compositions using kernel-based quantile estimation with online adaptation of the constant offset."

[18] A. Klein, F. Ishikawa, and S. Honiden, "Towards network-aware service composition in the cloud," in *Proceedings of the 21st international conference on World Wide Web*, ser. WWW '12, New York, NY, USA: ACM, 2012, pp. 959–968.